APPENDIX A

```
//Copyright Unisys 2001

#ifndef LARGEINT_H
#define LARGEINT_H

#include <iostream>
#include <string>
using namespace std;

//Redefine this if code is running on a non-8bit system like 2200
#ifndef BITS_PER_BYTE
        #define BITS_PER_BYTE 8
#endif

//Redefine this to change the storage type. Storage type must be an
//unsigned type.
//Intrinsic type int must be a multiple of INTNODE_TYPE and at least
//twice the size of INTNODE_TYPE
#ifndef INTNODE_TYPE
        #define INTNODE_TYPE unsigned short
#endif

#ifndef MAX
        #define MAX(a,b)        (((a) > (b)) ? (a) : (b))
#endif

#ifndef MIN
        #define MIN(a,b)        (((a) < (b)) ? (a) : (b))
#endif

/*----======~~~~~~~~~~~~~~~~""""""""""""""""""""""~~~~~~~~~~~~~~~~======----*\
        Class LargeInt
        A new datatype to hold signed integers of arbitrary size.

        1.      Created class datatype
        2.      Added conversions to and from all datatypes, including:
                a. bool (0 = false, else true)
                b. char (char and signed char are different types)
                c-j. signed/ unsigned char/ short/ int/ long
                k-l. float/ double/ long double
                m. const char * (character array/ string) for printing and
                storing.
                        The LargeInt can also be printed using the iostream << operator, and can
                be initialized with the iostream >> operator. It will print in the radix specified in
                the ostream flags. Binary printing is not supported by ostream, but you can still
                print in binary by setting the LargeIntBase to Bin and then calling (const char *)
                before operator<< (ie, cout << (const char *)MyLargeInt;).
        3.      Decided to only make a signed version, and not a complimentary unsigned
        version, because the difference only exists when you have fixed bit numbers, but LargeInt
        has infinite bit capability. Although both a positive and negative 0 can be stored, I make
        sure that there is never a negative zero.
        4.      Implement most integer based arithmetic, logic, and comparison operations,
        including:
```

a. + - * / % +(unary) -(unary)
b. += -= *= /= %= ++ --
c. & | ^ << >> ~
d. &= |= ^= <<= >>=
e. < > <= >= == != !

5. No functions have the virtual label in order to remove the extra level of indirection in the V-Table. Only add them if I decide to inherit more types.

6. Class behavior is undefined if there are more than 2^(sizeof(int)*BITS_PER_BYTE-1)-4 bits in the number. Ways around even this limit (though you'll run out of RAM before reaching it) involve using the STL array class instead of IntNode lists, and using LargeInt numbers internally. Both of which reduce the performance and speed of the datatype enough that it's not worth it.

7. The class functions are all made such that no datatype size information is hardcoded. Everything is determined at runtime/compiletime to allow the program to run on computers where int is not 32-bits (ie, 36 or 64-bit computers). The only thing that is hardcoded is BITS_PER_BYTE. This can be changed and the rest of the functions should still behave correctly (assuming BITS_PER_BYTE remains accurate). One restriction is that built-int datatypes (int, short, etc) much be a multiple of bytes (ie BITS_PER_DATATYPE % BITS_PER_BYTE = 0).

8. The library is OS independent, and adheres to the Standard C++ Library. Any half-decent compiler should be able to compile it. GCC < 3.0 is not a half-decent compiler, LargeInt will only compile on GCC >= 3.0.

9. Computations are performed using int type, while the values are stored in INTNODE_TYPE. To change the storage type, simply redefine INTNODE_TYPE. Intrinsic type int must be a multiple of INTNODE_TYPE and at least twice the size of INTNODE_TYPE. Also, INTNODE_TYPE must be an unsigned type.

10. The internal data structure is a bi-directional list of IntNodes. A Most Significant Bit and Least Significant Bit pointer are used to access it. These will be null if the number is 0. In order to greatly speed up the creation of these lists (especially with the heavy reliance on temporary instances), IntNode overloads the new and delete operators and stores a pool of previously used nodes. So new and delete calls will only take half a dozen operations, instead of having to go through malloc and free. This gives an experimental speedup of 5.3x.

11. In order for bitwise operations to make sense, I had to add in preliminary fixed bit support. If fixed bit is off, ~, Twos, and ^ will do nothing. & and | will only work if all operands are positive, otherwise they will do nothing. If FixedBit is On, The ~, Twos, and ^ operators will truncate/expand the inputs to the FixedBitLength, convert to unsignedness, perform the operation on that specified bit length, and then return the usual result. & and | will do that only if either operand is negative. On converting to (const char *), the result will be truncated/expanded to the desired bit length and converted to unsignedness.

12. In order to reduce the overhead of temporaries, I implemented "copy by stealing". With this, the IntNode list (the number) from a temporary is stolen O(1) instead of copied O(n).

End User Notes

1. LargeInt is a datatype. This means you use it just like you would any other datatype. It has a superset of "int" functionality. Anything you do with an int variable, you can do with a LargeInt variable.

2. Additions to "int" funtionality are Sign(), Twos(), and Divide() functions

3. Efficiency notes:
   a. Unary operations are optimized for efficiency and speed. ++X is faster than X++, both are faster than X += 1. Use ++X/--X whenever possible.
   b. if(X) and if(!X) are faster than comparing to zero.
   c. Construction is faster than assignment (it involves one less temporary copy).

d. Constructing from a decimal number (const char *) is slow. Use Hex, Oct, or Bin.

e. Outputting in decimal (const char *) is slow. Use Hex, Oct, or Bin. Specify ios::hex to the ostream class, or set the LargeIntBase flag and call (const char *) before ostream<<.

f. Division and multiplication are the slowest operations. But that's nothing new. Remember that division and modulo can be performed at the same time with Divide().

g. If a constant is used more than once, construct a constant LargeInt to hold it. Otherwise a LargeInt will be constructed from it anyway everytime the constant is used with a LargeInt expression.

h. Use (-) instead of Twos() for regular two's compliment, unless you desire an unsigned result. (-) is faster.

4. The only storage that maintains full precision is the character array. All other conversions are potentially lossy. The LargeInt can be serialized using ofstream << and initialized using ifstream >>, as well as from user input (o/istream, o/istrstream).

a. Note that using routines from math.h will cause potentially lossy conversions to take place. If you have to, convert to double, as this will retain the 53 most significant bits of a number 1023 bits long.

5. The library is not thread aware. It is up to the user to make sure a LargeInt is only accessed once at a time, the same as any other variable.

6. Changing the flags will affect all LargeInt variables.

a. Set/GetFlags() See documentation below.

b. Set/GetPrePostFix() See documentation below

c. You should push/pop any flags you change in a function to avoid unexpected side-effects in the calling function.

i. Example code:

```
foo()
{
        //Store old state info and set new info
        unsigned int OldFixedBit =
                GetFlag(LargeInt::FixedBit);
        SetFlag(LargeInt::FixedBit, 36);
        LargeInt::BaseEnum OldBase =
                GetFlag(LargeInt::PrintBase);
        SetFlag(LargeInt::PrintBase,
                LargeInt::Bin);
        //Store old pre/postfix for binary output //and set new
        string OldPrefix, OldPostfix;
        GetPrePostFix(OldPrefix, OldPostfix);
        SetPrePostFix("1\"", "\"");

        //Restore old state
        SetPrePostFix(OldPrefix, OldPostfix);
        SetFlag(LargeInt::PrintBase, OldBase);
        SetFlag(LargeInt::FixedBit, OldFixedBit);
}
```

ii. Note that PrintBase and Pre/Postfix only apply to (const char *) conversion, so you usually won't need to care about those.

7. Make sure LargeInt::FixedBit is set appropriately before performing operations on LargeInt variables.

a. If Fixed Bit is turned off

i. ~, ^, and Twos() will return 0 if FixedBit is turned off.

ii. & and | will return 0 if FixedBit is turned off and either operand is negative. If both operands are positive they will perform infinite precision & and |.

iii.LargeIntUnsigned will be turned off.

b. If Fixed Bit is turned on
i. ~, Twos(), ^, and (const char *) conversion will unsign/ truncate the result to the fixed bit length. & and | will do that if either operand is negative.
ii. Be aware of this if using | for quick addition. | is not significantly faster than + on LargeInts.
c.
8. Do not store the return value from (const char *). The return value will become an invalid pointer the next time (const char *) is called on any LargeInt. Use strcpy to save the result, or construct a C++ string or iostream off of it.

```
\*----+++++++,,,,,,,,,,,,,________________________,,,,,,,,,,,,,+++++++----*/
class LargeInt
{
protected:

//*** Internal Data Structures ***

        //List node structure (bi-directional)
        template<class T>
        struct _TIntNode
        {
                //Constructors
                _TIntNode();   //m_Value = 0;
                _TIntNode(const T &);  //m_Value = uint
                _TIntNode(const _TIntNode &);
                //The operator= only copies the value.
                //The next/prev pointers remain unchainged
                _TIntNode &operator=(const _TIntNode &);

                //Member Data.
                T m_Value;
                _TIntNode *m_pPrev;
                _TIntNode *m_pNext;
                typedef T Type;

                //Decrease the allocation/deallocation overhead of the
                //dynamic node list by providing a node pool to enable
                //reuse of node memory.

                //Points to the head of the pool list (singly-linked)
                static _TIntNode *m_pNodePool;
                //The number of currently existing owners. An owner is any
                //instance that could potentially use an IntNode. If no //more owners exist, we
                can free up all the memory in the //node pool.
                static unsigned int m_OwnerCount;
                //The number of nodes in the pool.
                static unsigned int m_NodeCount;
                //The number of nodes allowed in the pool per owner.
                //Increase the value to have fewer //allocations/deallocations. Decrease the
                value to conserve //more run-time memory.
                #define NODEPOOL_RATIO 4
                //Controll alloc/dealloc
                static void *operator new(unsigned int);
                static void operator delete(void *);
        };
```

```
public:
        //Flag Enumerations
        enum BaseEnum {Hex = 16, Dec = 10, Oct = 8, Bin = 2};
        enum OnOffEnum {On = 1, Off = 0};
        enum FlagEnum {PrintBase, FixedBit};

protected:
        //Local defines
        typedef _TIntNode<INTNODE_TYPE> IntNode;
        #define BITS_PER_INT (sizeof(unsigned int) * BITS_PER_BYTE)
        #define BITS_PER_NODE (sizeof(INTNODE_TYPE) * BITS_PER_BYTE)
        #define INT_MSB_MASK (1 << (BITS_PER_INT - 1))
        #define INTNODE_MSB_MASK (1 << (sizeof(INTNODE_TYPE) * BITS_PER_BYTE -
1))
        //This is simply the point at which I can no longer count the
        //number of bits a LargeInt is holding using an integer. Knowing
        //the exact number of bits a LargeInt holds is necessary for //several functions. Also,
        indexing is required for const char * //conversion.
        #define MAX_NODES ((((unsigned int)(-1) >> 1) - 4) / BITS_PER_NODE)

//*** Internal List Methods ***
        //The list will always be compacted. Which means any funtions
        //that might alter the list will call Compact before returning. //A LargeInt who's value is
        0 will have an empty list,
        //and will be positive.

        //The head (LSB) and tail (MSB) of the list.
        IntNode *m_pLSB, *m_pMSB;
        //Number of nodes.
        unsigned int m_NodeCount;

        //Insert a node (nodes) at the end with value = Value
        void Expand(unsigned int);
        void Expand(IntNode::Type = 0);

        //Remove nodes from the end that are no longer needed
        //Since compact must always be called anytime the list changes,
        //compact will take care of assuring there is never a negative
        //zero.
        void Compact();

        //Truncate to FixedBit # bits. Do nothing if FixedBit isn't
        //turned on.
        void Truncate();

//*** Internal Helper Functions ***

        //Helpers for parsing the char*. The first "unsigned int" is the
        //string length
        //The second "int" is the index into char *

        //Looks for + or -
        //Returns false if no more characters in the string.
        //Returns the index to the character after the sign in "int &"
        bool ParseSign(const char *, int, int &);
        //Looks for 0x, 0, h, x, o, or b (upper or lower case)
        //Sets BaseEnum to be Hex, Oct, Dec, or Bin based on the prefix
        //Returns false if no more characters in the string.
        bool ParseBase(const char *, int, int &, BaseEnum &);
        //Parses a hex number starting at index "int"
        //Returns the index to the first character not parsed.
        int ParseHex(const char *, int, int);
        int ParseDec(const char *, int, int);
        int ParseOct(const char *, int, int);
        int ParseBin(const char *, int, int);

        //Common code for constructing from integer
        //bool=true means signed, false means unsigned
        void FromInt(bool, unsigned int);
        //Common code for converting to integer
        unsigned int ToInt() const;
        //Common code for constructing from floating point
        void FromFloat(long double);
        //Common code for converting to floating point
        enum FloatEnum {Float, Double, LongDouble};
        long double ToFloat(FloatEnum) const;

        //Converts a signed LargeInt to a fixed bit unsigned number. This
        //result will only be used for printing purposes or as a
        //temporary in binary operations. It is no longer valid if //fixedbit changes, and cannot
        be returned to signed format.
        friend LargeInt UnSign(const LargeInt &);

//*** Flag Variables ***

        //false = positive, true = negative
        bool m_Sign;
        //false = regular signed int. true = this is the output of
        //UnSign(). It is only inherited if m_CopyByStealing is enabled.
        bool m_UnSigned;

        static unsigned int m_FixedBit;
        static BaseEnum m_Base;

        static string m_HexPrefix;
        static string m_HexPostfix;
        static string m_DecPrefix;
        static string m_DecPostfix;
        static string m_OctPrefix;
        static string m_OctPostfix;
        static string m_BinPrefix;
        static string m_BinPostfix;

        //Enabling this can vastly speed up copy construction, but should
        //only be used when you know you're copying a temporary which
        //will be destroyed shortly anyway. Otherwise you'll lose data.
        //false = perform full copy on each node
        //true = steal the nodes becuase it's just a temporary
        //This only affects the next call to operator=(const LargeInt &)
        //or LargeInt(const LargeInt&). They reset this flag to false.
        static bool m_CopyByStealing;

public:
```

```
//*** Constructors ***

        //Default, initialized pointers to NULL
        LargeInt();
        //Constructs large int from bool, signed/unsigned char,
        //short, int, and long
        LargeInt(signed int);
        LargeInt(unsigned int);
        LargeInt(signed long);
        LargeInt(unsigned long);
        //Constructs large int from float and double
        LargeInt(double);
        LargeInt(long double);
        //Constructs large int from constant (char array). The constant
        //will be assumed to be signed. The number will be extracted from //the MSB until the
        first formatting error or EOS.
        //Prefixes:
        //      The standard C prefixes ox(Hex) and o(Oct) will be
        //accepted, as well as h(hex), x(hex), o(oct), and b(bin) (upper
        //or lower case). The sign (-/+/nothing) must come before the //prefix. Converting from
        decimal is slow. Make your constants in //hex, oct, or bin format when possible.
        LargeInt(const char *);
        //Copy constructor
        LargeInt(const LargeInt &);

        //Destructor.
        ~LargeInt();

        //Assignment Operator. Everything is converted to LargeInt before
        //assignement is called.
        LargeInt &operator=(const LargeInt &);

//*** Flag Functions ***

        //Set/get the internal flags to value uint.
        //Supported flags are:
        //      LargeIntBase:   Hex, Dec, Oct, Bin      -       for (const
        //char *) conversion
        //      LargeIntFixedBit: Off, # - If Off, turns off FixedBit
        //operations.
        //                                                      - If #, turns on
        //FixedBit operations. Specifies # bits used for bitwise
        //operations (~, &, |, ^) and for conversion (to int, double, and
        //const char *).
        static void SetFlag(FlagEnum, unsigned int);
        static unsigned int GetFlag(FlagEnum);

        //Set/Get the prefix and postfix used for printing the LargeInt
        //in the specified base.
        //BaseEnum = Hex, Dec, Oct, Bin. Strings can be anything. const
        //char * will print (+/-)Prefix####Postfix. Default Prefixes and
        //postfixes are already defined.
        static void SetPrePostFix(const string &, const string &,
                BaseEnum = m_Base);
        static void GetPrePostFix(string &, string &, BaseEnum = m_Base);

        //Returns true if this is negative, else false;
        bool Sign();

//*** Conversions ***

        //Convert a LargeInt to bool, signed/unsigned char,
        //short, int, and long
        operator bool() const;
        operator char() const;
        operator signed char() const;
        operator unsigned char() const;
        operator signed short() const;
        operator unsigned short() const;
        operator signed int() const;
        operator unsigned int() const;
        operator signed long() const;
        operator unsigned long() const;
        //Convert a large int to float or double.
        operator float() const;
        operator double() const;
        operator long double() const;
        //Print the number in character format. Printing in decimal is
        //slow. Print in hex, oct, or bin if possible.
        //Do not store the return value. The return value will become an
        //invalid pointer the next time (const char *) is called on any
        //LargeInt. Use strcpy to save the result, or construct a C++
        //string or iostream off of it.
        operator const char *() const;

//*** Operator Macros ***
#define OPDEFN(Op, LType, RType)     friend LargeInt operator Op(LType,
        RType);
#define L const LargeInt &

//*** Arithmetic ***

        OPDEFN(+, L, L)                       OPDEFN(+, L, bool)
        OPDEFN(+, L, char)            OPDEFN(+, L, const char *)
        OPDEFN(+, L, signed char)     OPDEFN(+, L, unsigned char)          OPDEFN(+, L,
short)          OPDEFN(+, L, unsigned short)
        OPDEFN(+, L, signed int)      OPDEFN(+, L, unsigned int)           OPDEFN(+, L,
long)           OPDEFN(+, L, unsigned long)
        OPDEFN(+, L, float)           OPDEFN(+, L, double)                 OPDEFN(+, L,
long double)    OPDEFN(+, bool,L)                              OPDEFN(+, char,L)
                OPDEFN(+, const char *,       L)
        OPDEFN(+, signed char,L)      OPDEFN(+, unsigned, char,L)   OPDEFN(+, short,
        L)             OPDEFN(+, unsigned short,L)
        OPDEFN(+, signed int,L)               OPDEFN(+, unsigned int,       L)
        OPDEFN(+, long,L)                     OPDEFN(+, unsigned long,L)
        OPDEFN(+, float,              L)      OPDEFN(+, double,     L)
        OPDEFN(+, long double,        L)      OPDEFN(-, L, L)
        OPDEFN(-, L, bool)            OPDEFN(-, L, char)            OPDEFN(-, L, const
char *) OPDEFN(-, L, signed char)     OPDEFN(-, L, unsigned char)   OPDEFN(-, L, short)
                OPDEFN(-, L, unsigned short)  OPDEFN(-, L, signed int)        OPDEFN(-, L,
unsigned int)   OPDEFN(-, L, long)                    OPDEFN(-, L, unsigned long)
        OPDEFN(-, L, float)                   OPDEFN(-, L, double)          OPDEFN(-, L,
long double)
        OPDEFN(-, bool,L)                     OPDEFN(-, char,L)
        OPDEFN(-, const char *,        L)
        OPDEFN(-, signed char,L)   OPDEFN(-, unsigned char,      L)      OPDEFN(-,
short,L)        OPDEFN(-, unsigned short,     L)
        OPDEFN(-, signed int,L)              OPDEFN(-, unsigned int,           L)
        OPDEFN(-, long,L)                    OPDEFN(-, unsigned long,     L)
        OPDEFN(-, float,L)         OPDEFN(-, double,                  L)
        OPDEFN(-, long double,L)   OPDEFN(*, L, L)
        OPDEFN(*, L, bool)         OPDEFN(*, L, char)                 OPDEFN(*, L,
const char *)
        OPDEFN(*, L, signed char)  OPDEFN(*, L, unsigned char)        OPDEFN(*, L,
short)          OPDEFN(*, L, unsigned short)
        OPDEFN(*, L, signed int)   OPDEFN(*, L, unsigned int)         OPDEFN(*, L,
long)           OPDEFN(*, L, unsigned long)
        OPDEFN(*, L, float)        OPDEFN(*, L, double)               OPDEFN(*, L,
long double)
        OPDEFN(*, bool,L)                    OPDEFN(*, char,L)
        OPDEFN(*, const char *,        L)
        OPDEFN(*, signed char,       L)      OPDEFN(*, unsigned char,L)   OPDEFN(*,
short,  L)              OPDEFN(*, unsigned short,L)
        OPDEFN(*, signed int,L)              OPDEFN(*, unsigned int,      L)
        OPDEFN(*, long,L)                    OPDEFN(*, unsigned long,L)
        OPDEFN(*, float,L)         OPDEFN(*, double,       L)
        OPDEFN(*, long double,L)   OPDEFN(/, L, L)
        OPDEFN(/, L, bool)         OPDEFN(/, L, char)                 OPDEFN(/, L,
const char *)
        OPDEFN(/, L, signed char)  OPDEFN(/, L, unsigned char)        OPDEFN(/, L,
short)          OPDEFN(/, L, unsigned short)
        OPDEFN(/, L, signed int)   OPDEFN(/, L, unsigned int)         OPDEFN(/, L,
long)           OPDEFN(/, L, unsigned long)
        OPDEFN(/, L, float)        OPDEFN(/, L, double)               OPDEFN(/, L,
long double)    OPDEFN(/, bool,L)                         OPDEFN(/, char,L)
                OPDEFN(/, const char *,L)
        OPDEFN(/, signed char,L)   OPDEFN(/, unsigned char,L)   OPDEFN(/, short,L)
        OPDEFN(/, unsigned short,L)
        OPDEFN(/, signed int,L)              OPDEFN(/, unsigned int,      L)
        OPDEFN(/, long,L)                    OPDEFN(/, unsigned long,L)
        OPDEFN(/, float,L)         OPDEFN(/, double,       L)
        OPDEFN(/, long double,L)   OPDEFN(%, L, L)
        OPDEFN(%, L, bool)         OPDEFN(%, L, char)           OPDEFN(%, L, const
char *) OPDEFN(%, L, signed char)  OPDEFN(%, L, unsigned char)  OPDEFN(%, L, short)
                OPDEFN(%, L, unsigned short) OPDEFN(%, L, signed int)     OPDEFN(%, L,
unsigned int)   OPDEFN(%, L, long)                   OPDEFN(%, L, unsigned long)
        OPDEFN(%, L, float)                  OPDEFN(%, L, double)         OPDEFN(%, L,
long double)
        OPDEFN(%, bool,L)                    OPDEFN(%, char,L)
        OPDEFN(%, const char *,L)  OPDEFN(%, signed char,        L)      OPDEFN(%,
unsigned char,L)        OPDEFN(%, short,L)
        OPDEFN(%, unsigned short,L) OPDEFN(%, signed int,L)
        OPDEFN(%, unsigned int,      L)      OPDEFN(%, long,L)
        OPDEFN(%, unsigned long,L)  OPDEFN(%, float,L)
        OPDEFN(%, double,      L)            OPDEFN(%, long double,        L)

        LargeInt &operator +=(const LargeInt &);
        LargeInt &operator -=(const LargeInt &);
        LargeInt &operator *=(const LargeInt &);
        LargeInt &operator /=(const LargeInt &);
        LargeInt &operator %=(const LargeInt &);
```

```
            //Use this to get the result and remainder at the same time,
            //since they are found concurrently in the divide operation. This
            //uses a fast, recursive, divide and conquer algorithm I devised.
            friend LargeInt Divide(const LargeInt &, const LargeInt &,
                    LargeInt &);

    //*** Unary ***
            //These operations are all optimized for speed and efficiency.

            friend LargeInt operator +(const LargeInt &);
            friend LargeInt operator -(const LargeInt &);
            LargeInt &operator ++();//prefix. Prefix is faster than
                                          //postfix. Use this instead when possible.
            LargeInt operator ++(int);         //postfix.
            LargeInt &operator --();
            LargeInt operator --(int);
            //One's compliment.
            friend LargeInt operator ~(const LargeInt &);
            //Two's compliment. ++(~X) will only work as long as you remain
            //in the same Fixed Bit Length, since ++ isn't a bitwise
            //operator. The (-) operator is also the equivalent of Two's
            //compliment, except that it keeps the number in signed format.
            //This function returns an unsigned fixed bit result instead.
            friend LargeInt Twos(const LargeInt &);

    //*** Binary ***

            OPDEFN(&, L, L)                      OPDEFN(&, L, bool)
            OPDEFN(&, L, char)           OPDEFN(&, L, const char *)
            OPDEFN(&, L, signed char)    OPDEFN(&, L, unsigned char)        OPDEFN(&, L,
    short)          OPDEFN(&, L, unsigned short)
            OPDEFN(&, L, signed int)     OPDEFN(&, L, unsigned int)         OPDEFN(&, L,
    long)           OPDEFN(&, L, unsigned long)
            OPDEFN(&, L, float)          OPDEFN(&, L, double)               OPDEFN(&, L,
    long double)    OPDEFN(&, bool,L)                               OPDEFN(&, char,L)
                    OPDEFN(&, const char *,        L)
            OPDEFN(&, signed char,L)     OPDEFN(&, unsigned char,L)  OPDEFN(&, short,L)
            OPDEFN(&, unsigned short,L)
            OPDEFN(&, signed int,L)              OPDEFN(&, unsigned int,L)      OPDEFN(&,
    long,L)                 OPDEFN(&, unsigned long,L)
            OPDEFN(&, float,L)           OPDEFN(&, double,L)
            OPDEFN(&, long double,L)     OPDEFN(|, L, L)
            OPDEFN(|, L, bool)           OPDEFN(|, L, char)                 OPDEFN(|, L,
    const char *)   OPDEFN(|, L, signed char)    OPDEFN(|, L, unsigned char)   OPDEFN(|, L,
    short)                  OPDEFN(|, L, unsigned short)  OPDEFN(|, L, signed int)
            OPDEFN(|, L, unsigned int)   OPDEFN(|, L, long)                 OPDEFN(|, L,
    unsigned long)  OPDEFN(|, L, float)                  OPDEFN(|, L, double)
            OPDEFN(|, L, long double)
            OPDEFN(|, bool,L)                    OPDEFN(|, char,L)
            OPDEFN(|, const char *,L)    OPDEFN(|, signed char,L)      OPDEFN(|, unsigned
    char,L) OPDEFN(|, short,L)
            OPDEFN(|, unsigned short,L)  OPDEFN(|, signed int,L)
            OPDEFN(|, unsigned int,L)    OPDEFN(|, long,L)
            OPDEFN(|, unsigned long,L)   OPDEFN(|, float,L)
            OPDEFN(|, double,L)          OPDEFN(|, long double,L)
```

```
        OPDEFN(^, L, L)                        OPDEFN(^, L, bool)
        OPDEFN(^, L, char)             OPDEFN(^, L, const char *)
        OPDEFN(^, L, signed char)      OPDEFN(^, L, unsigned char)             OPDEFN(^, L,
short)          OPDEFN(^, L, unsigned short)
        OPDEFN(^, L, signed int)       OPDEFN(^, L, unsigned int)              OPDEFN(^, L,
long)           OPDEFN(^, L, unsigned long)
        OPDEFN(^, L, float)            OPDEFN(^, L, double)                    OPDEFN(^, L,
long double)    OPDEFN(^, bool,L)                                    OPDEFN(^, char,L)
                OPDEFN(^, const char *,L)
        OPDEFN(^, signed char,L)       OPDEFN(^, unsigned char,L)    OPDEFN(^, short,
        L)              OPDEFN(^, unsigned short,L)
        OPDEFN(^, signed int,L)                OPDEFN(^, unsigned int,         L)
        OPDEFN(^, long,L)                      OPDEFN(^, unsigned long,L)
        OPDEFN(^, float,L)             OPDEFN(^, double,       L)
        OPDEFN(^, long double,L)       OPDEFN(<<, L, L)                        OPDEFN(<<, L,
bool)           OPDEFN(<<, L, char)            OPDEFN(<<, L, const char *)     OPDEFN(<<, L,
signed char)    OPDEFN(<<, L, unsigned char) OPDEFN(<<, L, short)              OPDEFN(<<, L,
unsigned short) OPDEFN(<<, L, signed int)      OPDEFN(<<, L, unsigned int)     OPDEFN(<<, L,
long)                   OPDEFN(<<, L, unsigned long) OPDEFN(<<, L, float)
        OPDEFN(<<, L, double)                  OPDEFN(<<, L, long double)
        OPDEFN(<<, bool,L)             OPDEFN(<<, char,L)
        OPDEFN(<<, const char *,L)     OPDEFN(<<, signed char,L)     OPDEFN(<<, unsigned
char,L) OPDEFN(<<, short,      L)
        OPDEFN(<<, unsigned short,L) OPDEFN(<<, signed int,L)        OPDEFN(<<, unsigned
int,L)  OPDEFN(<<, long,L)
        OPDEFN(<<, unsigned long,L)    OPDEFN(<<, float,L)
        OPDEFN(<<, double,L)           OPDEFN(<<, long double, L)
        OPDEFN(>>, L, L)                       OPDEFN(>>, L, bool)
        OPDEFN(>>, L, char)            OPDEFN(>>, L, const char *)
        OPDEFN(>>, L, signed char)     OPDEFN(>>, L, unsigned char) OPDEFN(>>, L, short)
        OPDEFN(>>, L, unsigned short)
        OPDEFN(>>, L, signed int)      OPDEFN(>>, L, unsigned int)             OPDEFN(>>, L,
long)           OPDEFN(>>, L, unsigned long)
        OPDEFN(>>, L, float)           OPDEFN(>>, L, double)
        OPDEFN(>>, L, long double)     OPDEFN(>>, bool,L)
        OPDEFN(>>, char,L)             OPDEFN(>>, const char *,        L)
        OPDEFN(>>, signed char,        L)      OPDEFN(>>, unsigned char,       L)
        OPDEFN(>>, short,              L)      OPDEFN(>>, unsigned short,      L)
        OPDEFN(>>, signed int,         L)      OPDEFN(>>, unsigned int,        L)
        OPDEFN(>>, long,               L)      OPDEFN(>>, unsigned long,       L)
        OPDEFN(>>, float,              L)      OPDEFN(>>, double,              L)
        OPDEFN(>>, long double, L)

        LargeInt &operator &=(const LargeInt &);
        LargeInt &operator |=(const LargeInt &);
        LargeInt &operator ^=(const LargeInt &);
        LargeInt &operator <<=(const LargeInt &);
        LargeInt &operator >>=(const LargeInt &);

//*** Comparision ***
#undef OPDEFN
#define OPDEFN(Op, LType, RType)friend bool operator Op(LType, RType);

        OPDEFN(<, L, L)                        OPDEFN(<, L, bool)
        OPDEFN(<, L, char)             OPDEFN(<, L, const char *)
        OPDEFN(<, L, signed char)      OPDEFN(<, L, unsigned char)             OPDEFN(<, L,
short)          OPDEFN(<, L, unsigned short)
```

```
        OPDEFN(<, L, signed int)        OPDEFN(<, L, unsigned int)          OPDEFN(<, L,
long)           OPDEFN(<, L, unsigned long)
        OPDEFN(<, L, float)             OPDEFN(<, L, double)                OPDEFN(<, L,
long double)    OPDEFN(<, bool,         L)                          OPDEFN(<, char,
        L)              OPDEFN(<, const char *,                 L)
        OPDEFN(<, signed char,          L)      OPDEFN(<, unsigned char,    L)
        OPDEFN(<, short,                L)      OPDEFN(<, unsigned short,   L)
        OPDEFN(<, signed int,  L)       OPDEFN(<, unsigned int,             L)
        OPDEFN(<, long,                 L)      OPDEFN(<, unsigned long,    L)
        OPDEFN(<, float,                L)      OPDEFN(<, double,                   L)
        OPDEFN(<, long double,          L)      OPDEFN(>, L, L)
        OPDEFN(>, L, bool)              OPDEFN(>, L, char)                  OPDEFN(>, L,
const char *)   OPDEFN(>, L, signed char)       OPDEFN(>, L, unsigned char) OPDEFN(>, L,
short)                  OPDEFN(>, L, unsigned short) OPDEFN(>, L, signed int)
        OPDEFN(>, L, unsigned int)      OPDEFN(>, L, long)                  OPDEFN(>, L,
unsigned long)  OPDEFN(>, L, float)                     OPDEFN(>, L, double)
        OPDEFN(>, L, long double)
        OPDEFN(>, bool,L)                               OPDEFN(>, char,L)
        OPDEFN(>, const char *,         L)      OPDEFN(>, signed char,      L)
        OPDEFN(>, unsigned char,L)      OPDEFN(>, short,L)
        OPDEFN(>, unsigned short,L)     OPDEFN(>, signed int,L)
        OPDEFN(>, unsigned int,L)       OPDEFN(>, long,L)
        OPDEFN(>, unsigned long,L)      OPDEFN(>, float,L)
        OPDEFN(>, double,       L)              OPDEFN(>, long double,      L)
        OPDEFN(<=, L, L)                        OPDEFN(<=, L, bool)
        OPDEFN(<=, L, char)             OPDEFN(<=, L, const char *)
        OPDEFN(<=, L, signed char)      OPDEFN(<=, L, unsigned char) OPDEFN(<=, L, short)
        OPDEFN(<=, L, unsigned short)
        OPDEFN(<=, L, signed int)       OPDEFN(<=, L, unsigned int)         OPDEFN(<=, L,
long)           OPDEFN(<=, L, unsigned long)
        OPDEFN(<=, L, float)            OPDEFN(<=, L, double)
        OPDEFN(<=, L, long double)      OPDEFN(<=, bool,            L)
        OPDEFN(<=, char,                L)      OPDEFN(<=, const char *,    L)
        OPDEFN(<=, signed char,         L)      OPDEFN(<=, unsigned char,   L)
        OPDEFN(<=, short,               L)      OPDEFN(<=, unsigned short,  L)
        OPDEFN(<=, signed int,          L)      OPDEFN(<=, unsigned int,    L)
        OPDEFN(<=, long,                L)      OPDEFN(<=, unsigned long,   L)
        OPDEFN(<=, float,               L)      OPDEFN(<=, double,          L)
        OPDEFN(<=, long double, L)      OPDEFN(>=, L, L)                    OPDEFN(>=, L,
bool)           OPDEFN(>=, L, char)                     OPDEFN(>=, L, const char *)
        OPDEFN(>=, L, signed char)      OPDEFN(>=, L, unsigned char) OPDEFN(>=, L, short)
        OPDEFN(>=, L, unsigned short)OPDEFN(>=, L, signed int)      OPDEFN(>=, L,
unsigned int)   OPDEFN(>=, L, long)                     OPDEFN(>=, L, unsigned long)
        OPDEFN(>=, L, float)            OPDEFN(>=, L, double)               OPDEFN(>=, L,
long double)
        OPDEFN(>=, bool,                L)      OPDEFN(>=, char,            L)
        OPDEFN(>=, const char *,L)      OPDEFN(>=, signed char,     L)      OPDEFN(>=,
unsigned char,L)        OPDEFN(>=, short,               L)      OPDEFN(>=, unsigned short,L)
        OPDEFN(>=, signed int,          L)      OPDEFN(>=, unsigned int,L)  OPDEFN(>=,
long,           L)      OPDEFN(>=, unsigned long,L) OPDEFN(>=, float,               L)
        OPDEFN(>=, double,L)            OPDEFN(>=, long double, L)
        OPDEFN(==, L, L)                        OPDEFN(==, L, bool)
        OPDEFN(==, L, char)             OPDEFN(==, L, const char *)
        OPDEFN(==, L, signed char)      OPDEFN(==, L, unsigned char) OPDEFN(==, L, short)
        OPDEFN(==, L, unsigned short)
        OPDEFN(==, L, signed int)       OPDEFN(==, L, unsigned int)         OPDEFN(==, L,
long)           OPDEFN(==, L, unsigned long)
```

```
        OPDEFN(==, L, float)          OPDEFN(==, L, double)
        OPDEFN(==, L, long double)    OPDEFN(==, bool,      L)
        OPDEFN(==, char,              L)    OPDEFN(==, const char *,      L)
        OPDEFN(==, signed char,       L)    OPDEFN(==, unsigned char,     L)
        OPDEFN(==, short,             L)    OPDEFN(==, unsigned short,    L)
        OPDEFN(==, signed int,        L)    OPDEFN(==, unsigned int,      L)
        OPDEFN(==, long,              L)    OPDEFN(==, unsigned long,     L)
        OPDEFN(==, float,             L)    OPDEFN(==, double,    L)      OPDEFN(==,
long double, L) OPDEFN(!=, L, L)                 OPDEFN(!=, L, bool)
        OPDEFN(!=, L, char)                 OPDEFN(!=, L, const char *)   OPDEFN(!=, L,
signed char)    OPDEFN(!=, L, unsigned char) OPDEFN(!=, L, short)         OPDEFN(!=, L,
unsigned short) OPDEFN(!=, L, signed int)   OPDEFN(!=, L, unsigned int)   OPDEFN(!=, L,
long)                OPDEFN(!=, L, unsigned long) OPDEFN(!=, L, float)
        OPDEFN(!=, L, double)         OPDEFN(!=, L, long double)
        OPDEFN(!=, bool,              L)    OPDEFN(!=, char,              L)
        OPDEFN(!=, const char *,L)    OPDEFN(!=, signed char,      L)      OPDEFN(!=,
unsigned char,L)     OPDEFN(!=, short,            L)      OPDEFN(!=, unsigned short,L)
        OPDEFN(!=, signed int, L)     OPDEFN(!=, unsigned int,L)    OPDEFN(!=, long,
        L)      OPDEFN(!=, unsigned long,L)  OPDEFN(!=, float,               L)
        OPDEFN(!=, double,L)          OPDEFN(!=, long double, L)
        OPDEFN(&&, L, L)                    OPDEFN(&&, L, bool)
        OPDEFN(&&, L, char)           OPDEFN(&&, L, const char *)
        OPDEFN(&&, L, signed char)    OPDEFN(&&, L, unsigned char) OPDEFN(&&, L, short)
        OPDEFN(&&, L, unsigned short)
        OPDEFN(&&, L, signed int)     OPDEFN(&&, L, unsigned int)          OPDEFN(&&,
L, long)        OPDEFN(&&, L, unsigned long)
        OPDEFN(&&, L, float)          OPDEFN(&&, L, double)
        OPDEFN(&&, L, long double)    OPDEFN(&&, bool,             L)
        OPDEFN(&&, char,              L)    OPDEFN(&&, const char *,      L)
        OPDEFN(&&, signed char,       L)    OPDEFN(&&, unsigned char,     L)
        OPDEFN(&&, short,             L)    OPDEFN(&&, unsigned short,    L)
        OPDEFN(&&, signed int,        L)    OPDEFN(&&, unsigned int,      L)
        OPDEFN(&&, long,              L)    OPDEFN(&&, unsigned long,     L)
        OPDEFN(&&, float,             L)    OPDEFN(&&, double,            L)
        OPDEFN(&&, long double, L)    OPDEFN(||, L, L)                    OPDEFN(||, L,
bool)           OPDEFN(||, L, char)                 OPDEFN(||, L, const char *)
        OPDEFN(||, L, signed char)    OPDEFN(||, L, unsigned char)  OPDEFN(||, L, short)
        OPDEFN(||, L, unsigned short) OPDEFN(||, L, signed int)     OPDEFN(||, L,
unsigned int)   OPDEFN(||, L, long)                 OPDEFN(||, L, unsigned long)
        OPDEFN(||, L, float)          OPDEFN(||, L, double)         OPDEFN(||, L, long
double)
        OPDEFN(||, bool,              L)    OPDEFN(||, char,              L)
        OPDEFN(||, const char *,L)    OPDEFN(||, signed char,      L)      OPDEFN(||,
unsigned char,L)     OPDEFN(||, short,            L)      OPDEFN(||, unsigned short,L)
        OPDEFN(||, signed int,L)      OPDEFN(||, unsigned int,L)    OPDEFN(||, long,
        L)
        OPDEFN(||, unsigned long,L)   OPDEFN(||, float,      L)
        OPDEFN(||, double,      L)    OPDEFN(||, long double, L)

        //Use if(X) or if(!X) instead of comparing to zero.
        friend bool operator !(const LargeInt &);

#undef OPDEFN
#undef L

//*** Stream Operators ***
```

```
        //Help the largeint to be streamed as output just like an int
        friend std::ostream &operator <<
                (std::ostream &, const LargeInt &);
        //Help intput to be streamed into the LargeInt
        friend std::istream &operator >>
                (std::istream &, LargeInt &);
};

#ifdef LARGEINTLIBRARY_BUILD
        #include "LargeIntNode.tcpp"
#endif

#endif
```